6.001 SICP
Variations on a Scheme

Beyond Scheme – designing language variants:
Lazy evaluation
• Complete conversion – normal order evaluator
• Upward compatible extension – lazy, lazy-memo

Evaluation model

Rules of evaluation:
• If expression is self-evaluating (e.g. a number), just return value
• If expression is a name, look up value associated with that name in environment
• If expression is a lambda, create procedure and return
• If expression is special form (e.g. if) follow specific rules for evaluating subexpressions
• If expression is a compound expression
  • Evaluate subexpressions in any order
  • If first subexpression is primitive (or built-in) procedure, just apply it to values of other subexpressions
  • If first subexpression is compound procedure (created by lambda), evaluate the body of the procedure in a new environment, which extends the environment of the procedure with a new frame in which the procedure’s parameters are bound to the supplied arguments

Alternative models for computation

• Applicative Order:
  • evaluate all arguments, then apply operator

• Normal Order:
  • go ahead and apply operator with unevaluated argument subexpressions
  • evaluate a subexpression only when value is needed
  • by primitive procedure (that is, primitive procedures are "strict" in their arguments)

Normal Order Example

(begin (write-line "eval arg") 222)
=> (begin (write-line "eval arg") (+ 222 222))
=> 444

Normal order (lazy evaluation) versus applicative order

• How can we change our evaluator to use normal order?
  • Create “delayed objects” – expressions whose evaluation has been deferred
  • Change the evaluator to force evaluation only when needed
• Why is normal order useful?
  • What kinds of computations does it make easier?
How can we implement lazy evaluation?

(include (l-apply procedure arguments env) ; changed
  (cond ((primitive-procedure? procedure)
    (apply-primitive-procedure
      procedure
      (list-of-arg-values arguments env)))
  ((compound-procedure? procedure)
    (l-eval-sequence
      (procedure-body procedure)
      (extend-environment
        (procedure-parameters procedure)
        (list-of-delayed-args arguments env)
        (procedure-environment procedure)))
  (else (error "Unknown proc" procedure))))

Lazy Evaluation — l-eval

• Most of the work is in l-apply; need to call it with:
  • actual value for the operator
  • just expressions for the operands
  • the environment...

(define (l-eval exp env)
  (cond ((self-evaluating? exp) exp)
    ...
    (else (error "Unknown expression" exp))))

Meval versus L-Eval

(define (meval exp env)
  (cond ((self-evaluating? exp) exp)
    ...
    (else (error "Unknown expression type -- EVAL" exp))))

(define (l-eval exp env)
  (cond ((self-evaluating? exp) exp)
    ...
    (else (error "Unknown expression" exp))))

Actual vs. Delayed Values

(define (actual-value exp env)
  (force-it (l-eval exp env)))

(define (list-of-delayed-args exps env)
  (if (no-operands? exps) '()
    (cons (delay-it (first-operand exps) env)
      (list-of-delayed-args (rest-operands exps) env))))

(define (list-of-arg-values exps env)
  (if (no-operands? exps) '()
    (cons (actual-value (first-operand exps) env)
      (list-of-arg-values (rest-operands exps) env))))

Representing Thunks

• Abstractly — a thunk is a "promise" to return a value when later needed ("forced")

• Concretely — our representation:

Thunks – delay-it and force-it

(define (delay-it exp env) (list 'thunk exp env))
(define (thunk? obj) (tagged-list? obj 'thunk))
(define (thunk-exp thunk) (cadr thunk))
(define (thunk-env thunk) (caddr thunk))

(define (force-it obj)
  (cond ((thunk? obj)
    (actual-value (thunk-exp obj) (thunk-env obj)))
    (else obj)))

(define (actual-value exp env)
  (force-it (l-eval exp env)))
Memo-izing evaluation

- In lazy evaluation, if we reuse an argument, have to reevaluate each time
- In normal evaluation, argument is evaluated once, and just referenced
- Can we keep track of values once we’ve obtained them, and avoid cost of reevaluation?

Memo-izing Thunks

- Idea: once thunk exp has been evaluated, remember it
- If value is needed again, just return it rather than recompute

Concretely – mutate a thunk into an evaluated-thunk

Thunks – Memoizing Implementation

```
(define (evaluated-thunk? obj)
  (tagged-list? obj 'evaluated-thunk))
(define (thunk-value evaluated-thunk)
  (cadr evaluated-thunk))

(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (actual-value (thunk-exp obj) (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)
           (set-cdr! (cdr obj) '())
           result))
       ((evaluated-thunk? obj) (thunk-value obj))
       (else obj)))
```

Lazy Evaluation – other changes needed

- Example – need actual predicate value in conditional if...
  (define (l-eval-if exp env)
    (if (true? (actual-value (if-predicate exp) env))
        (l-eval (if-consequent exp) env)
        (l-eval (if-alternative exp) env)))
- Example – don’t need actual value in assignment...
  (define (l-eval-assignment exp env)
    (set-variable-value!
      (assignment-variable exp)
      (l-eval (assignment-value exp) env)
      env)
    'ok)

Laziness and Language Design

- We have a dilemma with lazy evaluation
  - Advantage: only do work when value actually needed
  - Disadvantages
    – not sure when expression will be evaluated; can be very big issue in a language with side effects
    – may evaluate same expression more than once
- Memoization doesn’t fully resolve our dilemma
  - Advantage: Evaluate expression at most once
  - Disadvantage: What if we want evaluation on each use?
- Alternative approach: give programmer control!

Variable Declarations: lazy and lazy-memo

- Handle lazy and lazy-memo extensions in an upward-compatible fashion:
  (lambda (a (b lazy) c (d lazy-memo)) ...)
Syntax Extensions – Parameter Declarations

```scheme
(define (first-variable var-decls) (car var-decls))
(define (rest-variables var-decls) (cdr var-decls))
(define declaration? pair?)
(define (parameter-name var-decl)
  (if (pair? var-decl) (car var-decl) var-decl))
(define (lazy? var-decl)
  (and (pair? var-decl) (eq? 'lazy (cadr var-decl))))
(define (memo? var-decl)
  (and (pair? var-decl)
       (eq? 'lazy-memo (cadr var-decl))))
```

Controllably Memo-izing Thunks

- thunk – never gets memoized
- thunk-memo – first eval is remembered
- evaluated-thunk – memoized-thunk that has already been evaluated

```
(thunk-memo exp env)
(evaluate-thunk result when forced)
```

A new version of delay-it

- Look at the variable declaration to do the right thing...

```scheme
(define (delay-it decl exp env)
  (cond ((not (declaration? decl))
         (l-eval exp env))
        ((lazy? decl)
         (list 'thunk exp env))
        ((memo? decl)
         (list 'thunk-memo exp env))
        (else (error "unknown declaration:" decl))))
```

Change to force-it

```scheme
(define (force-it obj)
  (cond ((thunk? obj) ; eval, but don’t remember it
         (actual-value (thunk-exp obj) (thunk-env obj)))
        ((memoized-thunk? obj) ; eval and remember
         (let ((result
                 (actual-value (thunk-exp obj) (thunk-env obj))))
            (set-car! obj 'evaluated-thunk)
            (set-car! (cdr obj) result)
            (set-cdr! (cdr obj) '())
            result))
        ((evaluated-thunk? obj) (thunk-value obj))
        (else obj)))
```

Changes to l-apply

- Key: in l-apply, only delay "lazy" or "lazy-memo" params
  - make thunks for "lazy" parameters
  - make memoized-thunks for "lazy-memo" parameters

```scheme
(define (l-apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
         ...) ; as before; apply on list-of-arg-values
        ((compound-procedure? procedure)
         (l-eval-sequence
          (procedure-body procedure)
          (let ((params (procedure-parameters procedure))
                (extend-environment
                 (map parameter-name params)
                 (list-of-delayed-args params arguments env)
                 (procedure-environment procedure)))))
         (else (error "unknown proc" procedure))))
```

Deciding when to evaluate an argument...

- Process each variable declaration together with application subexpressions – delay as necessary:

```scheme
(define (list-of-delayed-args var-decls exps env)
  (if (no-operands? exps)
      '()
      (cons (delay-it (first-variable var-decls)
                      (first-operand exps) env)
            (list-of-delayed-args
             (rest-variables var-decls)
             (rest-operands exps) env))))
```
Summary

• Lazy evaluation – control over evaluation models
  • Convert entire language to normal order
  • Upward compatible extension
    – lazy & lazy-memo parameter declarations

Streams – a different way of structuring computation

• Imagine simulating the motion of an object
  • Use state variables, clock, equations of motion to update
  • State of the simulation captured in instantaneous values of state variables

Streams – a different way of structuring computation

• OR – have each object output a continuous stream of information
  • State of the simulation captured in the history (or stream) of values

How do we use this new lazy evaluation?

• Our users could implement a stream abstraction:
  (define (cons-stream x (y lazy-memo))
    (lambda (msg)
      (cond ((eq? msg 'stream-car) x)
            ((eq? msg 'stream-cdr) y)
            (else (error "unknown stream msg" msg)))))

Stream Object

• A pair-like object, except the cdr part is lazy
  (not evaluated until needed):

  • Example
    (define x (cons-stream 99 (/ 1 0)))
    (stream-car x) => 99
    (stream-cdr x) => error - divide by zero

Decoupling computation from description

• Can separate order of events in computer from apparent order of events in procedure description

  (define (stream-interval a b)
    (if (> a b)
      the-empty-stream
      (cons-stream a (stream-interval (+ a 1) b)))))
Some details on stream procedures

\[
\text{(define (stream-filter pred str)} \\
\text{\quad (if (pred (stream-car str)))} \\
\text{\quad \quad (cons-stream (stream-car str))} \\
\text{\quad \quad (stream-filter pred (stream-cdr str))))}
\]

Decoupling order of evaluation

\[
\text{(stream-filter prime? (str-in 1 10000000))}
\]

Result: Infinite Data Structures!

- Some very interesting behavior
  \[
  \text{(define ones (cons-stream 1 ones))} \\
  \text{(stream-car (stream-cdr ones)) \Rightarrow 1}
  \]
  The infinite stream of 1's!
  \[
  \text{ones: 1 1 1 1 1 1 ....}
  \]
- Compare:
  \[
  \text{(define ones (cons 1 ones)) \Rightarrow error, ones undefined}
  \]

Finding all the primes

\[
\begin{array}{cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
Streams Programming

- Signal processing:
  \[ x[n] \rightarrow \text{Delay} \rightarrow y[n] \]
- Streams model:
  \[ x \rightarrow \text{add-streams} \rightarrow \text{stream-scale} \rightarrow y \]

Integration as an example

```scheme
(define (integral integrand init dt)
  (define int
    (cons-stream init
      (add-streams (stream-scale dt integrand)
                   int)))
  int)

(integral ones 0 2)
=> 0 2 4 6 8
```

Summary

- We can control when arguments are evaluated
  - By making a lazy evaluator
  - By changing the evaluator to allow specification of arguments
- Changing the evaluator requires a small amount of work but dramatically shifts the behavior of the system
  - Applicative order versus Normal order
- Using a lazy evaluator lets us separate the apparent order of computation inherent in a problem from the actual order of evaluation inside the machine

Stages of an interpreter

Lexical analyzer

Parser

Evaluator

Environment

Printer

"(average 4 (+ 5 5))"

```
(average 4 (+ 5 5))
(average 4 (+ 5 5))
```

"7"